Тема

«ClassLoaders»

Состав

Виды загрузчиков

Процесс работы загрузчика

Модель безопасности Java

Виды загрузчиков

Любой класс (экземпляр класса java.lang.Class в среде и .class файл в файловой системе), используемый в среде исполнения был так или иначе загружен каким-либо загрузчиком в Java.
Для того, чтобы получить загрузчик, которым был загружен класс A, необходимо воспользоваться методом A.class.getClassLoader().
Классы загружаются по мере надобности, за небольшим исключением. Некоторые базовые классы из rt.jar (java.lang.* в частности) загружаются при старте приложения. Классы расширений ($JAVA_HOME/lib/ext), пользовательские и большинство системных классов загружаются по мере их использования.
Различают 3-и вида загрузчиков в Java. Это — базовый загрузчик (bootstrap), системный загрузчик (System Classloader), загрузчик расширений (Extension Classloader).
Bootstrap — реализован на уровне JVM и не имеет обратной связи со средой исполнения. Данным загрузчиком загружаются классы из директории $JAVA_HOME/lib. Т.е. всеми любимый rt.jar загружается именно базовым загрузчиком. Поэтому, попытка получения загрузчика у классов java.* всегда заканчивается null'ом. Это объясняется тем, что все базовые классы загружены базовым загрузчиком, доступа к которому из управляемой среды нет.
Управлять загрузкой базовых классов можно с помощью ключа -Xbootclasspath, который позволяет переопределять наборы базовых классов.
System Classloader — системный загрузчик, реализованный уже на уровне JRE. В Sun JRE — это класс sun.misc.Launcher$AppClassLoader. Этим загрузчиком загружаются классы, пути к которым указаны в переменной окружения CLASSPATH.
Управлять загрузкой системных классов можно с помощью ключа -classpath или системной опцией java.class.path.
Extension Classloader — загрузчик расширений. Данный загрузчик загружает классы из директории $JAVA_HOME/lib/ext. В Sun JRE — это класс sun.misc.Launcher$ExtClassLoader.
Управлять загрузкой расширений можно с помощью системной опции java.ext.dirs.
Различают текущий загрузчик (Current Classloader) и загрузчик контекста (Context Classloader).
Current Classloader — это загрузчик класса, код которого в данный момент исполняется. Текущий загрузчик используется по умолчанию для загрузки классов в процессе исполнения. В часности, при использовании метода Class.forName("")/ClassLoader.loadClass("") или при любой декларации класса, ранее не загруженного.
Context Classloader — загрузчик контекста текущего потока. Получить и установить данный загрузчик можно с помощью методов Thread.getContextClassLoader()/Thread.setContextClassLoader(). Загрузчик контекста устанавливается автоматически для каждого нового потока. При этом, используется загрузчик родительского потока.

Процесс работы загрузчика

Право загрузки класса рекурсивно делегируется от самого нижнего загрузчика в иерархии к самому верхнему. Такой подход позволяет загружать классы тем загрузчиком, который максимально близко находится к базовому. Так достигается максимальная область видимости классов. Под областью видимости подразумевается следующее. Каждый загрузчик ведет учет классов, которые были им загружены. Множество этих классов и назвается областью видимости.
Рассмотрим процесс загрузки более детально. Пусть в систем исполнения встретилась декларация переменной пользовательского класс Student.
1) Системный загрузчик попытается поискать в кеше класс Student.
_1.1) Если класс найден, загрузка окончена.
_1.2) Если класс не найден, загрузка делегируется загрузчику расширений.
2) Загрузчик расширений попытается поискать в кеше класс Student.
_2.1) Если класс найден, загрузка окончена.
_2.2) Если класс не найден, загрузка делегируется базовому загрузчику.
3) Базовый загрузчик попытается поискать в кеше класс Student.
_3.1) Если класс найден, загрузка окончена.
_3.2) Если класс не найден, базовый загрузчик попытается его загрузить.
__3.2.1) Если загрузка прошла успешно, она закончена ;)
__3.2.2) Иначе управление предается загрузчику раширений.
_3.3) Загрузчик расширений пытается загрузить класс.
__3.3.1) Если загрузка прошла успешно, она закончена ;)
__3.3.2) Иначе управление предается системному загрузчику.
_3.4) Системный загрузчик пытается загрузить класс.
__3.4.1) Если загрузка прошла успешно, она закончена ;)
__3.4.2) Иначе генерируется исключение java.lang.ClassNotFoundException.

Если в системе присутствуют пользовательские загрузчики, они должны
а) расширять класс java.lang.ClassLoader;
б) поддерживать модель динамической загрузки.

Реализация

Определим интерфейс модулей. Пусть модуль сначала загружается (load), потом исполняется (run), возвращая результат и затем уже выгружается (unload). Данный код представляет собой API для разработки модулей. Его можно скомпилировать отдельно и упаковать в *.jar для поставки отдельно от основного приложения

```
public interface Module {

  public static final int EXIT_SUCCESS = 0;
  public static final int EXIT_FAILURE = 1;

  public void load();
  public int run();
  public void unload();

}

* This source code was highlighted with Source Code Highlighter.
```

Рассмотрим реализацию загрузчика модулей. Данный загрузчик загружает код классов из определенной директории, путь к которой указан в переменной pathtobin.

```
import java.io.File;
import java.io.FileInputStream;
```

```
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStream;

public class ModuleLoader extends ClassLoader {

  /**
   * Путь до директории с модулями.
   */
  private String pathtobin;

  public ModuleLoader(String pathtobin, ClassLoader parent) {
    super(parent);
    this.pathtobin = pathtobin;
  }

  @Override
  public Class<?> findClass(String className) throws
ClassNotFoundException {
    try {
      /**
       * Получем байт-код из файла и загружаем класс в рантайм
       */
      byte b[] = fetchClassFromFS(pathtobin + className + ".class");
      return defineClass(className, b, 0, b.length);
    } catch (FileNotFoundException ex) {
      return super.findClass(className);
    } catch (IOException ex) {
      return super.findClass(className);
    }

  }

  /**
   * Взято из www.java-tips.org/java-se-tips/java.io/reading-a-file-into-
a-byte-array.html
   */
  private byte[] fetchClassFromFS(String path) throws
FileNotFoundException, IOException {
    InputStream is = new FileInputStream(new File(path));

    // Get the size of the file
    long length = new File(path).length();

    if (length > Integer.MAX_VALUE) {
      // File is too large
    }

    // Create the byte array to hold the data
    byte[] bytes = new byte[(int)length];

    // Read in the bytes
    int offset = 0;
    int numRead = 0;
    while (offset < bytes.length
        && (numRead=is.read(bytes, offset, bytes.length-offset)) >= 0) {
```

```
      offset += numRead;
    }

    // Ensure all the bytes have been read in
    if (offset < bytes.length) {
      throw new IOException("Could not completely read file "+path);
    }

    // Close the input stream and return bytes
    is.close();
    return bytes;

  }
}

* This source code was highlighted with Source Code Highlighter.
```

Теперь рассмотрим реализацию движка загрузки модулей. Директория с модулями (файлами .class) указывается в качестве параметра приложению.

```
import java.io.File;

public class ModuleEngine {

  public static void main(String args[]) {
    String modulePath = args[0];
    /**
     * Создаем загрузчик модулей.
     */
    ModuleLoader loader = new ModuleLoader(modulePath,
ClassLoader.getSystemClassLoader());

    /**
     * Получаем список доступных модулей.
     */
    File dir = new File(modulePath);
    String[] modules = dir.list();

    /**
     * Загружаем и исполняем каждый модуль.
     */
    for (String module: modules) {
      try {
        String moduleName = module.split(".class")[0];
        Class clazz = loader.loadClass(moduleName);
        Module execute = (Module) clazz.newInstance();

        execute.load();
        execute.run();
        execute.unload();

      } catch (ClassNotFoundException e) {
        e.printStackTrace();
      } catch (InstantiationException e) {
        e.printStackTrace();
      } catch (IllegalAccessException e) {
        e.printStackTrace();
```

```
      }
    }


  }

}

* This source code was highlighted with Source Code Highlighter.
```

Реализуем простейший модуль, который просто печатает на стандартный вывод информацию о стадиях своего исполнения. Это можно сделать в отдельном приложении добавив к CLASSPATH путь до скомпилированного .jar файла с классом Module (API).

```
public class ModulePrinter implements Module {

  @Override
  public void load() {
    System.out.println("Module " + this.getClass() + " loading ...");
  }

  @Override
  public int run() {
    System.out.println("Module " + this.getClass() + " running ...");
    return Module.EXIT_SUCCESS;
  }

  @Override
  public void unload() {
    System.out.println("Module " + this.getClass() + "
inloading ...");
  }
}

* This source code was highlighted with Source Code Highlighter.
```

Скомпилировав данный код, результат в виде одного class файла можно скопировать в отдельную директорию, путь к которой необходимо указать в качестве параметра основного приложения.

Модель безопасности Java

Как известно, в виртуальной машине класс определяется своим полным именем и экземпляром ClassLoader’а которым он был загружен. Так, для VM классы с абсолютно одинаковыми именами, загруженные разными ClassLoader’ами будут разными: к примеру, попытка привести один из объект такого класса к другому обречена на провал.

Перед тем, как обсуждать безопасность необходимо рассмотреть несколько базовых понятий из этой области.

В модели безопасности Java с любым классом кроме ClassLoader’а связана еще одна сущность: *code base*. Попросту – местро откуда класс был загружен (URL ресурса или ссылка на локальный ресурс). Кроме того, у класса может быть *сертификат* – “честное слово” какой-нибудь организации или человека, что код класса будет вести себя хорошо. Code location и сертификаты (если есть) вместе образуют *code source*.

Давайте посмотрим на code source какого-нибудь класса. Для этого достаточно выполнить вот такой-вот фрагмент кода:

```
        System.out.println(Test.class.getProtectionDomain().getCodeSource());
```
В моем случае я вижу такие строки:

```
(file:/C:/apps/eclipse/workspace/Test/bin/ )
```
Permission – (разрешение, право) это класс, который идентифицирует доступность какого-нибудь действия. К примеру FilePermission – контролирует доступность операций над частью файловой системы. Вот простой пример:

```
        Permission p = new FilePermission("temp/*", "read");
```
Созданный объект – разрешение на чтение (но не на запись) любых файлов в дериктории temp.

Как видно, permission состоит из двух частей – имени (в FilePermission имя – это путь к папке) и списка действий.

Code Source и коллекция разрешений – образуют домен безопасности (protection domain). С каждым классом ассоциируется домен безопасности (не каждый класс, правда, сможет его посмотреть).

Последний из ключевых классов модели – SecurityManager. Его задача – давать окончательный ответ на вопрос “а стоит ли разрешить действие”, которое пытается выполнить класс.

Типичный пример использования SecurityManager’а выглядит так:

```
SecurityManager security = System.getSecurityManager();
if (security != null)
        security.checkPermission(permToCheck);
```
Метод checkPermission отбросит SecurityException, если операция запрещена.

Есть несколько нюансов, которые необходимо понимать при использовании SecurityManager’а. Самый основной: безопасность проверяется в некотором контексте. Попробую объяснить простыми словами: SecurityManager проверяет не просто доступность действия, он исследует все классы из текущего стека на предмет соответствия политике безопасности. Если хотя-бы один из классов не имеет достаточных прав – будет выброшен SecurityException.

Для тех кто любит вникать в детали: SecurityManager, естественно, не занимается проверкой безопасности единолично. Для проверки всего контекста используется AccessController, который в свою очередь передает работу AccessControllerContext. AccessControllerContext инкапсулирует контекст вызова (стек) и именно он проводит окончательную проверку:

```
for (int i=0; i< context.length; i++) {
    if (context[i] != null &&  !context[i].implies(perm)) {
  ...
   throw new AccessControlException("access denied "+perm, perm);
    }
}
```
Последнее о чем нужно знать на этом этапе – понятие “политики безопасности”. Политика безопасности сопостовляет Code Source’ы (что за код выполяется) Principal’ы (кто выполняет код) с наборами разрешений. В Java политика реализуется при помощи классов наследников java.security.Policy (сам класс – абстрактный). В JDK есть только один наследник Policy: PolicyFile. Как и следует из названия PolicyFile в качестве источника информации о правах использует файл. Файл с описанием политики по умолчанию находится в JRE_HOME/lib/security/java.policy. Описание формата файла выходит за рамки этой заметки. Его легко можно найти в сети.

В самом простом варианте, файл состоит из блоков grant, в которых описаны разрешения в формате
permission <Класс Permission> “<имя Permission’а>”, “<значение>”

К примеру:

```
grant {
  // allows anyone to listen on un-privileged ports
  permission java.net.SocketPermission "localhost:1024-", "listen";
};
```
Такой блок позволяет “слушать” порты от 1024-го и выше. Соответственно, порт 125 “услышать” не выйдет.

Теперь, когда основные понятия благополучно введены, самое время немного позабавиться “практикой”.
Попробуем выполнить действие, которое должно быть “запрещено”, и посмотрим, что будет.
Итак, поскольку код выше я скопировал прямо из java.policy, Security Maneger не должен позволить открыть порт 125.

```
try {
  new ServerSocket(124);
} catch (IOException e) {
  System.out.println("Could not listen on port: 124");
  System.exit(-1);
}
System.out.println("Listening OK.");
```
Код выполняется нормально. Как такое может быть? Ведь в файле с описанием политики явно сказано “можно слушать от 1024-го порта и выше”. Нюанс, который в свое время убил мне пару десятков нервных клеток: по умолчанию, в Java SE не установлен Securitiy Manager.
Соответственно, вообще никакие проверки не производятся. “Включить” Security Manager можно двумя способами.

```
1. System.setSecurityManager(new SecurityManager());
2. java -Djava.security.manager -Djava.security.policy==someURL SomeApp
```
(второй параметр можно опустить, он указывает расположение Policy файла).

Когда SecurityManager установлен, код не выполняется: SecurityException подробно объясняет нам, почему именно:

```
Exception in thread "main" java.security.AccessControlException: access
denied
 (java.net.SocketPermission localhost:124 listen,resolve)
```

Материалы для подготовки

1. https://habrahabr.ru/post/103830/
2. https://habrahabr.ru/post/104229/
3. http://voituk.kiev.ua/2008/08/18/bezopastnost-v-java/

Вопросы для самоконтроля

1) Для чего нужен SecurityManager в java?